



## 2.10 — Introduction to the preprocessor

👤 ALEX<sup>1</sup> ⌚ MARCH 22, 2024

When you compile your project, you might expect that the compiler compiles each code file exactly as you've written it. This actually isn't the case.

Instead, prior to compilation, each code (.cpp) file goes through a **preprocessing** phase. In this phase, a program called the **preprocessor** makes various changes to the text of the code file. The preprocessor does not actually modify the original code files in any way -- rather, all changes made by the preprocessor happen either temporarily in-memory or using temporary files.

### As an aside...

Historically, the preprocessor was a separate program from the compiler, but in modern compilers, the preprocessor may be built right into the compiler itself.

Most of what the preprocessor does is fairly uninteresting. For example, it strips out comments, and ensures each code file ends in a newline. However, the preprocessor does have one very important role: it is what processes `#include` directives (which we'll discuss more in a moment).

When the preprocessor has finished processing a code file, the result is called a **translation unit**. This translation unit is what is then compiled by the compiler.

### Related content

The entire process of preprocessing, compiling, and linking is called **translation**.

If you're curious, here is a list of [translation phases](https://en.cppreference.com/w/cpp/language/translation_phases) ([https://en.cppreference.com/w/cpp/language/translation\\_phases](https://en.cppreference.com/w/cpp/language/translation_phases))<sup>2</sup>. As of the time of writing, preprocessing encompasses phases 1 through 4, and compilation is phases 5 through 7.

## Preprocessor directives

When the preprocessor runs, it scans through the code file (from top to bottom), looking for preprocessor directives. **Preprocessor directives** (often just called *directives*) are instructions that start with a `#` symbol and end with a newline (NOT a semicolon). These directives tell the preprocessor to perform certain text manipulation tasks. Note that the preprocessor does not understand C++ syntax -- instead, the directives have their own syntax (which in some cases resembles C++ syntax, and in other cases, not so much).

In this lesson, we'll look at some of the most common preprocessor directives.

## As an aside...

Using `directives` (introduced in lesson [2.9 -- Naming collisions and an introduction to namespaces](https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/) (<https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/>)<sup>3</sup>) are not preprocessor directives (and thus are not processed by the preprocessor). So while the term `directive` *usually* means a `preprocessor directive`, this is not always the case.

## #Include

You've already seen the `#include` directive in action (generally to `#include <iostream>`). When you `#include` a file, the preprocessor replaces the `#include` directive with the contents of the included file. The included contents are then preprocessed (which may result in additional `#includes` being preprocessed recursively), then the rest of the file is preprocessed.

Consider the following program:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Hello, world!\n";
6 |     return 0;
7 | }
```

When the preprocessor runs on this program, the preprocessor will replace `#include <iostream>` with the contents of the file named "iostream" and then preprocess the included content and the rest of the file.

Once the preprocessor has finished processing the code file plus all of the `#included` content, the result is called a **translation unit**. The translation unit is what is sent to the compiler to be compiled.

## Key insight

A translation unit contains both the processed code from the code file, as well as the processed code from all of the `#included` files.

Since `#include` is almost exclusively used to include header files, we'll discuss `#include` in more detail in the next lesson (when we discuss header files).

## Macro defines

The `#define` directive can be used to create a macro. In C++, a **macro** is a rule that defines how input text is converted into replacement output text.

There are two basic types of macros: *object-like macros*, and *function-like macros*.

*Function-like macros* act like functions, and serve a similar purpose. Their use is generally considered unsafe, and almost anything they can do can be done by a normal function.

*Object-like macros* can be defined in one of two ways:

```
#define IDENTIFIER
#define IDENTIFIER substitution_text
```

The top definition has no substitution text, whereas the bottom one does. Because these are preprocessor directives (not statements), note that neither form ends with a semicolon.

The identifier for a macro uses the same naming rules as normal identifiers: they can use letters, numbers, and underscores, cannot start with a number, and should not start with an underscore. By convention, macro names are typically all upper-case, separated by underscores.

---

## Object-like macros with substitution text

When the preprocessor encounters this directive, any further occurrence of the identifier is replaced by *substitution\_text*. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

Consider the following program:

```
1 | #include <iostream>
2 |
3 | #define MY_NAME "Alex"
4 |
5 | int main()
6 | {
7 |     std::cout << "My name is: " << MY_NAME << '\n';
8 |
9 |     return 0;
10| }
```

The preprocessor converts the above into the following:

```
1 | // The contents of iostream are inserted here
2 |
3 | int main()
4 | {
5 |     std::cout << "My name is: " << "Alex" << '\n';
6 |
7 |     return 0;
8 | }
```

Which, when run, prints the output My name is: Alex.

Object-like macros with substitution text were used (in C) as a way to assign names to literals. This is no longer necessary, as better methods are available in C++. Object-like macros with substitution text should generally now only be seen in legacy code.

We recommend avoiding these kinds of macros altogether, as there are better ways to do this kind of thing. We discuss this more in lesson [5.1 -- Constant variables \(named constants\)](https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/) (<https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/>)<sup>4</sup>.

---

## Object-like macros without substitution text

*Object-like macros* can also be defined without substitution text.

For example:

```
1 | #define USE_YEN
```

Macros of this form work like you might expect: any further occurrence of the identifier is removed and replaced by nothing!

This might seem pretty useless, and it *is useless* for doing text substitution. However, that's not what this form of the directive is generally used for. We'll discuss the uses of this form in just a moment.

Unlike object-like macros with substitution text, macros of this form are generally considered acceptable to use.

---

## Conditional compilation

The *conditional compilation* preprocessor directives allow you to specify under what conditions something will or won't compile. There are quite a few different conditional compilation directives, but we'll only cover the three that are used by far the most here: *#ifdef*, *#ifndef*, and *#endif*.

The *#ifdef* preprocessor directive allows the preprocessor to check whether an identifier has been previously *#defined*. If so, the code between the *#ifdef* and matching *#endif* is compiled. If not, the code is ignored.

Consider the following program:

```
1 | #include <iostream>
2 |
3 | #define PRINT_JOE
4 |
5 | int main()
6 | {
7 |     #ifdef PRINT_JOE
8 |         std::cout << "Joe\n"; // will be compiled since PRINT_JOE is defined
9 |     #endif
10 |
11 |     #ifdef PRINT_BOB
12 |         std::cout << "Bob\n"; // will be excluded since PRINT_BOB is not defined
13 |     #endif
14 |
15 |     return 0;
16 | }
```

Because `PRINT_JOE` has been `#defined`, the line `std::cout << "Joe\n"` will be compiled.

Because `PRINT_BOB` has not been `#defined`, the line `std::cout << "Bob\n"` will be ignored.

`#ifndef` is the opposite of `#ifdef`, in that it allows you to check whether an identifier has *NOT* been `#defined` yet.

```

1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     #ifndef PRINT_BOB
6 |         std::cout << "Bob\n";
7 |     #endif
8 |
9 |     return 0;
10 | }
```

This program prints “Bob”, because `PRINT_BOB` was never `#defined`.

In place of `#ifdef PRINT_BOB` and `#ifndef PRINT_BOB`, you’ll also see `#if defined(PRINT_BOB)` and `#if !defined(PRINT_BOB)`. These do the same, but use a slightly more C++-style syntax.

## [\(\)#if 0](#) [🔗](#) [\(#if 0\)](#)<sup>5</sup>

One more common use of conditional compilation involves using `#if 0` to exclude a block of code from being compiled (as if it were inside a comment block):

```

1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Joe\n";
6 |
7 |     #if 0 // Don't compile anything starting here
8 |         std::cout << "Bob\n";
9 |         std::cout << "Steve\n";
10 |    #endif // until this point
11 |
12 |     return 0;
13 | }
```

The above code only prints “Joe”, because “Bob” and “Steve” are excluded from compilation by the `#if 0` preprocessor directive.

This provides a convenient way to “comment out” code that contains multi-line comments (which can’t be commented out using another multi-line comment due to multi-line comments being non-nestable):

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Joe\n";
6 |
7 |     #if 0 // Don't compile anything starting here
8 |         std::cout << "Bob\n";
9 |         /* Some
10 |          * multi-line
11 |          * comment here
12 |          */
13 |         std::cout << "Steve\n";
14 |     #endif // until this point
15 |
16 |     return 0;
17 | }
```

To temporarily re-enable code that has been wrapped in an `#if 0`, you can change the `#if 0` to `#if 1`:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "Joe\n";
6 |
7 |     #if 1 // always true, so the following code will be compiled
8 |         std::cout << "Bob\n";
9 |         /* Some
10 |          * multi-line
11 |          * comment here
12 |          */
13 |         std::cout << "Steve\n";
14 |     #endif
15 |
16 |     return 0;
17 | }
```

## Object-like macros don't affect other preprocessor directives

Now you might be wondering:

```
1 | #define PRINT_JOE
2 |
3 | #ifdef PRINT_JOE
4 |     // ...
```

Since we defined `PRINT_JOE` to be nothing, how come the preprocessor didn't replace `PRINT_JOE` in `#ifdef PRINT_JOE` with nothing?

Macros only cause text substitution for non-preprocessor commands. Since `#ifdef PRINT_JOE` is a preprocessor command, text substitution does not apply to `PRINT_JOE` within this command.

As another example:

```
1 | #define F00 9 // Here's a macro substitution
2 |
3 | #ifdef F00 // This F00 does not get replaced because it's part of another
4 | preprocessor directive
5 |     std::cout << F00 << '\n'; // This F00 gets replaced with 9 because it's
   | part of the normal code
   | #endif
```

However, the final output of the preprocessor contains no directives at all -- they are all resolved/stripped out before compilation, because the compiler wouldn't know what to do with them.

---

## The scope of #defines

Directives are resolved before compilation, from top to bottom on a file-by-file basis.

Consider the following program:

```
1 | #include <iostream>
2 |
3 | void foo()
4 | {
5 |     #define MY_NAME "Alex"
6 | }
7 |
8 | int main()
9 | {
10 |     std::cout << "My name is: " << MY_NAME << '\n';
11 |
12 |     return 0;
13 | }
```

Even though it looks like `#define MY_NAME "Alex"` is defined inside function `foo`, the preprocessor won't notice, as it doesn't understand C++ concepts like functions. Therefore, this program behaves identically to one where `#define MY_NAME "Alex"` was defined either before or immediately after function `foo`. For readability, you'll generally want to `#define` identifiers outside of functions.

Once the preprocessor has finished, all defined identifiers from that file are discarded. This means that directives are only valid from the point of definition to the end of the file in which they are defined. Directives defined in one code file do not have impact on other code files in the same project.

Consider the following example:

function.cpp:

```
1 | #include <iostream>
2 |
3 | void doSomething()
4 | {
5 |     #ifdef PRINT
6 |         std::cout << "Printing!\n";
7 |     #endif
8 |     #ifndef PRINT
9 |         std::cout << "Not printing!\n";
10 |    #endif
11 | }
```

main.cpp:

```
1 | void doSomething(); // forward declaration for function doSomething()
2 |
3 | #define PRINT
4 |
5 | int main()
6 | {
7 |     doSomething();
8 |
9 |     return 0;
10 | }
```

The above program will print:

Not printing!

Even though `PRINT` was defined in *main.cpp*, that doesn't have any impact on any of the code in *function.cpp* (`PRINT` is only `#defined` from the point of definition to the end of *main.cpp*). This will be of consequence when we discuss header guards in a future lesson.



**[Next lesson](#)**

2.11 [Header files](#)

6



**[Back to table of contents](#)**

7



**[Previous lesson](#)**

2.9 [Naming collisions and an introduction to namespaces](#)

3

8



[B](#) [U](#) [URL](#) [INLINE CODE](#) [C++ CODE BLOCK](#) [HELP!](#)

Leave a comment...

Name\*

@ Email\*



Notify me about replies:



POST COMMENT

Find a mistake? Leave a comment above!

 Avatars from <https://gravatar.com/><sup>10</sup> are connected to your provided email address.

462 COMMENTS

Newest ▼

**Nima**

April 19, 2024 2:16 pm

Thanks a lot for your amazing tutorials.

Just little fixes:

In places of `#ifdef PRINT_BOB` and `#ifndef PRINT_BOB`, you'll also see `#if defined(PRINT_BOB)` and `#if !defined(PRINT_BOB)`. These do the same, but use a slightly more C++ style syntax."

-----

Therefore, this program behaves identically to one where `#define MY_NAME "Alex"` was defined either before or immediately after **the** function `foo`. For readability, you'll generally want to `#define` identifiers outside of functions.

Last edited 5 days ago by Nima

0

Reply

**A W**

April 18, 2024 5:20 pm

In C, the pre-processor step basically takes the entire library of functions and inserts their declarations to the top of the file, and your code gets added to the bottom.

Is this the case in C++ as well?

 0 Reply**Alex** Author Reply to [A W](#)<sup>11</sup>  April 20, 2024 11:27 am

I don't understand your description of the C pre-processor. Are you talking about how `#includes` get resolved? If so, `#includes` get replaced by the content of the included file at the point of the `#include`. Most often `#includes` are at the top of the file, but not always.

 1 Reply**dddddds** Reply to [Alex](#)<sup>12</sup>  April 21, 2024 7:36 am

op

 0 Reply**Baker** April 17, 2024 4:54 am

I don't get why `print` isn't `#defined` when `void` is called even though it is defined in `main.cpp`.

 0 Reply**Alex** Author Reply to [Baker](#)<sup>13</sup>  April 18, 2024 8:34 am

From the lesson: "`PRINT` is only `#defined` from the point of definition to the end of `main.cpp`"

The `doSomething()` function is defined in a different file, where `PRINT` hasn't been defined yet.

 0 Reply

**Neblinus**

🕒 March 22, 2024 7:02 am

Hello, is this right? (English isn't my first language, maybe I'm wrong).

"By convention, macro names are typically all upper-case, separated by underscores."

```
#define identifier substitution_text
```

Shouldn't `substitution_text` be `SUBSTITUTION_TEXT` to follow the convention?

👍 0

➡ Reply

**Alex**

Author

👤 Reply to [Neblinus](#)<sup>14</sup> 🕒 March 22, 2024 4:24 pm

It's actually the identifier I'm talking about here. I updated it to use IDENTIFIER.

👍 0

➡ Reply

**Neblinus**👤 Reply to [Alex](#)<sup>15</sup> 🕒 March 23, 2024 5:51 am

Oh, ok, thank you so much.

Also, many thanks to you for creating and maintaining this awesome content, I hope someday I can be a good C++ programmer.

👍 0

➡ Reply

**IceFloe**

🕒 February 20, 2024 11:47 pm

thanks for the lesson, of course, but I still don't understand what is the advantage of macros over namespaces or just commenting on the right lines of code? should each `#ifdef` and `#ifndef` end with `#endif`? What if `#define USERNAME` is defined inside the `#ifdef USERNAME` `#endif` macro? will it work? And finally, do all macros have to have names? (exception `#if 0`)?

🔗 Last edited 2 months ago by IceFloe

👍 0

➡ Reply

**Alex**

Author

👤 Reply to [IceFloe](#)<sup>16</sup> 🕒 February 22, 2024 2:18 pm

Each `#ifdef` or `#ifndef` needs to end in an `#endif`. You can `#define` things inside an `#ifdef`. All defined macros need a name, but `#if` does not because it's evaluating an expression, which could be a macro name or a value.



0



Reply

**Swaminathan**

🕒 January 22, 2024 8:52 am

Hello Alex, Can you add some real life use cases for pre processor directives? (#include is fine as we have been using them for sometime. But it would be helpful to know when these things actually prove to be useful and indispensable)

Thanks in advance



2



Reply

**Swaminathan**🗨️ Reply to [Swaminathan](#)<sup>17</sup> 🕒 January 23, 2024 5:22 am

Okay, i see a very important use-case in the coming lessons on “Header guard”. My question is answered..

I find myself doing this multiple times(i.e. asking a question and finding its answer myself when I read later lessons). Sorry for this, but honestly, I don't know how to approach it otherwise.

🔗 *Last edited 3 months ago by Swaminathan*



2



Reply

**Alex** Author🗨️ Reply to [Swaminathan](#)<sup>18</sup> 🕒 January 23, 2024 10:20 am

No worries. If you have a question, first try looking in the next lesson to see if it's answered there. Sometimes lessons are a continuation of the prior.



5



Reply

**JZan**

🕒 December 3, 2023 3:23 am

> Macros only cause text substitution for normal code. Other preprocessor commands are ignored.

I'm not clear on the wording here. If other preprocessor commands were ignored, why does this seem to work fine? (C++17 with g++, Linux/Windows)

```
1 | #define NOT 0
2 | #if NOT
3 |     std::cout << "This doesn't print!\n";
4 | #endif
5 |
6 | #define YEAH 1
7 | #if YEAH
8 |     std::cout << "This prints!\n";
9 | #endif
```



2



Reply

**oussama**Reply to [JZan](#)<sup>19</sup> December 14, 2023 12:05 am

I think `#if` will check for the value of the macro  
and `#ifdef` will check only if the macro is defined or not

```
#define NOT 0
#ifdef NOT
std::cout << "This print!\n";
#endif
```



4



Reply

**Alex** AuthorReply to [JZan](#)<sup>19</sup> December 3, 2023 5:12 pm

Updated wording to "Macros only cause text substitution for non-preprocessor commands."

In your example, `#if NOT` does not become `#if 0`. Rather, the preprocessor checks whether `NOT` has been previously defined, and if so, what the value is, and then determines what code should be included. The end-result is the same.



5



Reply

**Adunakhor**

November 22, 2023 10:52 am

Btw, at online compilers websites (godbolt, for example, under the "+" sign) you can observe what preprocessor does with your code. `#include <iostream>` actually includes 36000 lines of code (headers, idk what it is) which is nuts.



3



Reply

**Kratos**

🕒 October 30, 2023 9:10 pm

> "This means that directives are only valid from the point of definition to the end of the file in which they are defined. Directives defined in one code file do not have impact on other code files in the same project."

Is the scope for the macros the files or the translation unit?

👍 1

➡ Reply

**Alex**

Author

🗨 Reply to [Kratos](#)<sup>20</sup> 🕒 October 31, 2023 10:31 am

Generally when we say file, we mean translation unit since that is actually what gets compiled.

👍 4

➡ Reply

**Mou**

🕒 October 18, 2023 6:22 am

In the "Object-like macros don't affect other preprocessor directives" section, for confirmation: A definition will ignore other preprocessors and only affect regular syntax, but the reason `#define FOO(substitute the FOO variable with the value 9)` was able to affect the FOO variable in `#ifndef FOO` was because after the `#ifndef` passed the syntax inside it, it entered the normal code, and then the FOO variable was able to find the definition given to it on line 1, applying the substitution rule. Is that a correct analysis of what happened there?

👍 0

➡ Reply

**Mou**🗨 Reply to [Mou](#)<sup>21</sup> 🕒 October 18, 2023 7:15 am

When I run this by a file it doesn't substitute FOO with 9 like it is implied in the "Object-like macros don't affect other preprocessor directives" section it outputs only the 0 I returned to check if it ran. Now from the way I view it this makes sense because it is contained within the `#ifndef` in this context, so it is not included in normal code, by the section says it should output 9, am I missing something in my understanding?

```
1 | #include <iostream>
2 |
3 | #define F00 9 // A Macro Substitute
4 |
5 | #ifdef F00 //This F00 does not get used because it is part of
   | another preprocessor directive(line 3)
6 |
7 | int main()
8 | {
9 |     std::cout << F00 << '\n'; // This F00 gets replaced with 9
   | because it's part of teh normal code
10 |     return 0;
11 | }
12 | #endif
```

👍 0

➡ Reply

**Alex** Author🗨 Reply to Mou<sup>22</sup> ⌚ October 20, 2023 1:32 pm

I'm confused by what you wrote. F00 isn't replaced on line 5, but it is replaced on line 9, so this program outputs 9.

👍 2

➡ Reply

**Yopi**

⌚ October 2, 2023 8:03 am

Thanks for the lesson

👍 4

➡ Reply

**azizkurtariciniz**

⌚ September 20, 2023 6:15 am

If you guys are having difficulty to understand this topic, try to take notes in bullet point ways. This will help you properly grasp the concepts as you put more time on the pieces.

👍 0

➡ Reply

**Thank you.**

⌚ September 19, 2023 9:57 am

I am studying Scott Meyers book "effective C++" and in item2 he mentioned about preprocessors and "Prefer consts, enums, and inlines to #defines" and I came here to see what preprocessors are. Thanks for this comprehensive and great explanations!



0



Reply

**Raylen**

🕒 September 17, 2023 2:07 pm

Hi, this doesn't make sense to me?

What's the point of defining a macro when you could just define a boolean for conditional compilation or a string for text substitution?

Can someone please explain this?



0



Reply

**Raylen**🗨️ Reply to [Raylen](#)<sup>23</sup> 🕒 September 21, 2023 12:36 pm

I mean you could definitely do something like this, maybe I misunderstood what conditional compilation, but this is how i'd do it

```
1 | bool compile { true };  
2 |  
3 | if(compile == true) {  
4 |     // Do something  
5 | } else {  
6 |  
7 | }
```



2



Reply

**Jake**🗨️ Reply to [Raylen](#)<sup>24</sup> 🕒 September 22, 2023 4:39 am

There is a difference between conditionally including code in compilation and conditionally executing code.

The preprocessor, upon being given the directives in the above article, strips code before compilation; the code is not so much as seen by the compiler.



5



Reply

**whitenight**🗨️ Reply to [Jake](#)<sup>25</sup> 🕒 October 24, 2023 4:25 pm

Really helpful comment, man, thanks



1



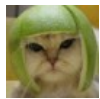
Reply



**Raylen**Reply to [Jake](#)<sup>25</sup> September 22, 2023 5:48 am

Ah I see, thank you I now understand.

1   Reply

**Alex** AuthorReply to [Raylen](#)<sup>23</sup> September 20, 2023 7:48 am

How would you define a boolean for conditional compilation?

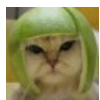
0   Reply

**Krishnakumar**

August 25, 2023 10:45 am

Did this lesson recently under a significant revision. It reads so much better now (or maybe because I have been reading these pages over the years and am able to better understand this).

0   Reply

**Alex** AuthorReply to [Krishnakumar](#)<sup>26</sup> August 28, 2023 2:17 pm

No significant revisions for this lesson in the last year. Just a bunch of minor edits for clarification/comprehensibility.

0   Reply

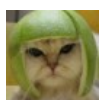
**Krishnakumar**

August 25, 2023 10:33 am

>There are quite a few different conditional compilation directives

Is there a comprehensive list somewhere?

1   Reply

**Alex** AuthorReply to [Krishnakumar](#)<sup>27</sup> August 28, 2023 2:13 pm

<https://en.cppreference.com/w/cpp/preprocessor/conditional>

3   Reply

**N T**

🕒 August 8, 2023 4:12 am

Are the contents of a header preprocessed before or after it is `#include`-ed?

The `#include` section of the lesson says, "The included contents are then preprocessed (along with the rest of the file), and then compiled", which suggests that the contents are preprocessed after inclusion. But the sentence after the code snippet says, "When the preprocessor runs on this program, the preprocessor will replace `#include <iostream>` with the preprocessed contents of the file named "iostream", which suggests the header's contents are preprocessed before inclusion.

👍 0

➡ Reply

**Alex**

Author

🗨 Reply to [N T](#)<sup>28</sup> 🕒 August 11, 2023 9:29 pm

I'm not 100% sure, but my understanding is that the content is included and then preprocessed. This makes sense to me, as you can `#define A`, then `#include` a header that has an `#ifdef A` inside, and it will consider A as defined.

👍 4

➡ Reply

**Helix**

🕒 June 27, 2023 6:37 am

Idk suddenly reading "In actuality" I was like, "Are we still doing C++?" :)

👍 2

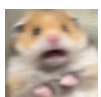
➡ Reply

**Emeka Daniel**🗨 Reply to [Helix](#)<sup>29</sup> 🕒 August 3, 2023 6:53 am

Hah!

👍 0

➡ Reply

**Zoltan**

🕒 June 18, 2023 1:31 pm

As a huge fan of this tutorial series I have to say, this was by far the most boring read so far in the course. Not to say it was not good, or that it's not important to cover.. it's well-written, but damn I almost fell asleep due to the topic at hand. :D

👍 19

➡ Reply

**Emeka Daniel**Reply to [Zoltan](#)<sup>30</sup> ⌚ August 3, 2023 6:56 am

We haven't gotten to the good part yet :).

👍 0    ➡ Reply

**Nobody**Reply to [Zoltan](#)<sup>30</sup> ⌚ July 1, 2023 7:04 am

I think its kind of interesting.  
I was hoping for a quiz..

👍 10    ➡ Reply

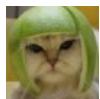
**Krishnakumar**

⌚ May 25, 2023 6:33 pm

>Historically, the preprocessor was a separate program from the compiler, but in modern compilers, the preprocessor is typically built right into the compiler itself.

The `cpp` command is still available as a standalone preprocessor on linux systems. I believe both `gcc` and `clang` invoke this during the compilation process?

👍 0    ➡ Reply

**Alex** AuthorReply to [Krishnakumar](#)<sup>31</sup> ⌚ May 29, 2023 11:08 pm

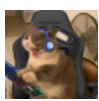
I think you are correct. Changed "is typically" to "may be".

👍 0    ➡ Reply

**Krishnakumar**Reply to [Alex](#)<sup>32</sup> ⌚ August 25, 2023 10:30 am

Thanks.


👍 0    ➡ Reply

**bigbawlsbobby69**

⌚ May 18, 2023 10:18 pm

What do you mean by saying that 'object-like macros' are dangerous? You make it sound like it's used to create malware or something.

Thanks for any help! :)

 Last edited 11 months ago by [bigbawlsbobby69](#)

 0  Reply



**Alex** Author

 Reply to [bigbawlsbobby69](#)<sup>33</sup>  May 19, 2023 10:08 am

Dangerous as in they can more easily lead to program errors and undefined behavior.

I replaced the word "dangerous" with "unsafe".

 0  Reply



**bigbawlsbobby69**

 Reply to [Alex](#)<sup>34</sup>  May 22, 2023 10:35 pm

Didn't initially put that together myself. My bad.

 0  Reply



**name**

 Reply to [Alex](#)<sup>34</sup>  May 20, 2023 2:44 pm

I like "dangerous" as it sounds more dramatic

 12  Reply



**Edson Freitas**

 May 12, 2023 9:30 am

Hi,

Congratulations by the efforts related to building this site. As a little improvement suggestion ...

In the frame "As an aside" whose inside text begins with "Using directives", I think this introductory sentence may mislead the reader to think the "Using" term not as a c++ instruction, but rather, as the english verb.

May I sugest 'The "using" c++ instruction' (double quotes or another way of differentiation intended) as an alternative introductory text?

Also, maybe the use of an anchor, directing the reader exactly to the point where the referenced page presents the "using" directive, can help improve the assimilation of the concept.

 0  Reply

**Alex** AuthorReply to [Edson Freitas](#)<sup>35</sup> May 14, 2023 7:27 pm

Using directives is already inside a code tag -- so it should be rendering differently than the surrounding text. Are you not seeing this? If not, what browser/OS are you on?

1

Reply

**Edson Freitas**Reply to [Alex](#)<sup>36</sup> May 15, 2023 10:44 pm

Wow!!!

I can see it now, but I can swear I couldn't see it before. It's a little subtle, but not that much. Are you sure it was already there before? Cause if yes, implies that I passed over it in at least two different times, on two different days.

Maybe I need more pills to sleep better, I don't deny it's possible, but ... are you really sure it was already there?

Long story. Call it beginning with mobbing at work, raised to more serious criminal levels over the years, raised to an ongoing criminal case since 2019, turned up to a ferocious fight for survival for both sides, and jail on the table for the losing side, death threats ... or you can just call it a titanic inductor of stress and fatigue, and a very, very strong resilience test.

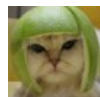
So I know I'm under the edge here, but to the point of missing it on two different times ... if true, it will raise a yellow flag.

In time ... it's firefox/OSX Catalina on mac mini (late 2012), and low light environment.

Well, at least half right ... rechecked, and I still think that the link (to naming-collisions-and-an-introduction-to-namespaces/) can be improved with an anchor pointing directly to the "using namespace std" section.

0

Reply

**Alex** AuthorReply to [Edson Freitas](#)<sup>37</sup> May 18, 2023 5:36 pm

Yup, I didn't change anything related to this. :)

0

Reply

**Edson Freitas**Reply to [Alex](#)<sup>38</sup> May 19, 2023 12:31 pm

While here I noticed other little clues arising again ... overdistraction, forgetting things where I shouldn't ... eh ... time to slow down and

rethink strategies ... again. Surprisingly, still optimistic ;-). One hell of a roadtrip ...

👍 0

➡ Reply



**J34NP3T3R**

🕒 May 11, 2023 5:51 pm

does this mean it could greatly increase the size of our executable when we include entire files when we just needed a few functions from those files ?

like when we include `iostream` to be able to use `std::cout` and that function only but then `#include <iostream>` appends into our code file the entire contents of `iostream` ?

👍 0

➡ Reply



**Alex** Author

👤 Reply to [J34NP3T3R](#)<sup>39</sup> 🕒 May 11, 2023 7:20 pm

Maybe or maybe not. If the standard library is linked in dynamically, then no, because the shared dynamic library is loaded at runtime. If the standard library is linked in statically, then possibly. But I'd expect a good linker to be able to exclude parts of the standard library that aren't actually used.

👍 1

➡ Reply



**Krishnakumar**

👤 Reply to [Alex](#)<sup>40</sup> 🕒 August 25, 2023 10:32 am

Is the standard library usually linked in dynamically on hosted systems?

👍 0

➡ Reply



**Alex** Author

👤 Reply to [Krishnakumar](#)<sup>41</sup> 🕒 August 28, 2023 2:11 pm

How the standard library is linked is up to the individual application.

👍 0

➡ Reply



**Krishnakumar**

👤 Reply to [Alex](#)<sup>42</sup> 🕒 October 10, 2023 12:12 pm

Thanks Alex. You mean dependent on the specific compilation flags passed, right? For `g++`, the relevant flag is `-static`.

👍 0

➡ Reply

**Alex** AuthorReply to [Krishnakumar](#)<sup>43</sup> ⌚ October 12, 2023 4:47 pm

Yeah, that's a better way of putting it.

👍 0

➡ Reply

**J34NP3T3R**Reply to [Alex](#)<sup>40</sup> ⌚ May 11, 2023 7:31 pm

ohh that is a relief to hear. thanks

👍 0

➡ Reply

**cPlusPlus\_Warlord**

⌚ April 30, 2023 1:31 pm

**Feature request:** you should implement a way to bookmark pages/lessons. This lesson is of particular interest to me, so it would help if there was a convenient way of returning to it in the future.

Nevertheless, another great lesson!

👍 4

➡ Reply

**Kenneth**

⌚ April 10, 2023 4:30 am

Im pretty sure I get the lesson but i dont really see the use of something being compiled/not compiled. Could you maybe provide an example?

👍 2

➡ Reply

**venu**Reply to [Kenneth](#)<sup>44</sup> ⌚ April 13, 2023 10:36 am

Lets say you have specific code which needs to included for some legacy versions

or

if you want to include few lines of code only for debugging

or

if want to compile few lines of code only for specific OS (like android or windows) or platform(x86,arm64 etc...)

📝 Last edited 1 year ago by venu

👍 5

➡ Reply

**Tim**Reply to [venu](#)<sup>45</sup> September 29, 2023 12:50 pm

I was wondering when I would even use anything taught in this lesson (except if 0 and if 1 for debugging) but that is a great example why I would. Thanks!



1

Reply

**Suku\_go**

March 27, 2023 3:55 am

What is `#endif` short hand for ? Do we really need `#ifndef` if we already got `#endif`. They seem to have the same behavior



0

Reply

**Alex**

Author

Reply to [Suku\\_go](#)<sup>46</sup> March 29, 2023 10:29 pm

`#ifdef` / `#ifndef` is short for "if defined" / "if not defined" and denotes where the text associated with an `#if`-based preprocessor command starts.

`#endif` is short for "end if", and denotes where the text associated with the prior `#if`-based preprocessor command ends.

We use `#ifndef` when we want to include a block of text when some symbol is not defined. The actual text to be included in such a case is the text between the `#ifndef` command and the `#endif` command.



3

Reply

**Talha Khan**

March 23, 2023 8:34 am

Probably a stupid tip but `#if 1` lets you print the block of code.



0

Reply

**Alex**

Author

Reply to [Talha Khan](#)<sup>47</sup> March 24, 2023 7:55 pm

Not print the block of code, but include it for compilation. I added a note about this to the lesson, because it's a useful way to temporarily include code that was previously `#if 0` for compilation.



3

Reply



**Krishnakumar**Reply to [Alex](#)<sup>48</sup> ⌚ August 25, 2023 10:36 am

Not just `0` and `1`, but we have a whole host of true-ish & false-ish constructs for conditional compilation.

```
#if 0, #if false, #if some_random_text
```

 disable the block of code until  

```
#endif
```

```
if true, #if 1, #if 2
```

 & other numeric values enable the block of code until  

```
#endif
```

✎ Last edited 7 months ago by Krishnakumar

👍 0

➡ Reply

**Jourvence**

⌚ March 10, 2023 10:57 am

Thanks for this Alex :)

👍 0

➡ Reply

**yellowEmu**

⌚ February 6, 2023 3:31 am

I have a question about printing object-like macros as string.

Let's say instead of `#define TEXT "SomeText"` I had `#define TEXT SomeText`

How would I correctly print `SomeText` ?

This doesn't work:

```
1 | #include <iostream>
2 |
3 | #define TEXT SomeText
4 |
5 | int main()
6 | {
7 |     std::cout << "TEXT" << std::endl;
8 | }
```

This prints `TEXT`

which confuses me since I thought the preprocessor would just "mindlessly" replace all occurrences of the string `TEXT` with `SomeText`, but apparently it doesn't.

The reason I am asking this is because I would like to print a macro that contains a path-string that is set by the `-D` command passed to the `gcc` command and that macro doesn't include



0

Reply

**yellowEmu**Reply to [yellowEmu](#)<sup>49</sup> February 6, 2023 4:01 am

I searched a little more for an answer and this post seems to have a good answer:

<https://stackoverflow.com/questions/71671400/print-a-define-macro-using-stdcout>  
(<https://stackoverflow.com/questions/71671400/print-a-define-macro-using-stdcout>)<sup>50</sup>

Apparently, because the preprocessor will NOT touch string literals (learned something new) there is an "idiom" called "[stringification](https://gcc.gnu.org/onlinedocs/gcc-3.4.3/cpp/Stringification.html)" (<https://gcc.gnu.org/onlinedocs/gcc-3.4.3/cpp/Stringification.html>)<sup>51</sup> that works like this (at least for gcc)

```
1 #define xstr(x) str(x)
2 #define str(x) #x
3 #define _TEST_ test
4 #include <iostream>
5
6 int main()
7 {
8     std::cout << xstr(_TEST_) << std::endl;
9 }
```

Last edited 1 year ago by yellowEmu

0

Reply

**Alex** AuthorReply to [yellowEmu](#)<sup>52</sup> February 6, 2023 11:16 am

Yup. The preprocessor doesn't understand C++ syntax, but that doesn't mean it's completely brainless. As you've discovered, the # symbol can be used to do stringifying.

For readers who want to know more, see <https://gcc.gnu.org/onlinedocs/gcc-4.8.5/cpp/Stringification.html>

4

Reply

**Krishnakumar**Reply to [Alex](#)<sup>53</sup> August 25, 2023 10:41 am

Interestingly "stringification" is now referred to by the gcc manuals as "stringizing". See the updated URL for the latest manual page on it: <https://gcc.gnu.org/onlinedocs/gcc-13.2.0/cpp/Stringizing.html>

👍 0    ➡ Reply



**Krishnakumar**

🗨 Reply to [Alex](#)<sup>53</sup> ⌚ May 25, 2023 6:46 pm

will this be explained in the text of these lessons somewhere?

👍 0    ➡ Reply



**Alex** Author

🗨 Reply to [Krishnakumar](#)<sup>54</sup> ⌚ May 29, 2023 11:09 pm

Wasn't planning on it.

👍 0    ➡ Reply



**Krishnakumar**

🗨 Reply to [Alex](#)<sup>55</sup> ⌚ August 25, 2023 10:38 am

That's a disappointment. Stringification looks quite cool.

👍 0    ➡ Reply



**Krishnakumar**

🗨 Reply to [Krishnakumar](#)<sup>56</sup> ⌚ November 16, 2023 4:45 am

Or stringizing, rather.

👍 0    ➡ Reply



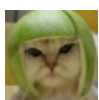
**Chien**

⌚ January 31, 2023 3:21 am

This also provides a convenient way to “comment out” code that contains multi-line comments (which can’t be commented out using another multi-line comment due to multi-line comments being non-nestable):

I'm confuse with this statement for #if 0  
Can you explain.

👍 0    ➡ Reply



**Alex** Author

🗨 Reply to [Chien](#)<sup>57</sup> ⌚ February 3, 2023 10:59 am

`#if 0` causes the preprocessor to exclude all of the code between this directive and the subsequent `#endif` for compilation.

Therefore, anything within these bounds is effectively ignored, which is essentially what commenting out your code does.

👍 2    ➡ Reply



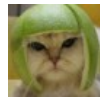
**Aersuy**

🗨 Reply to [Alex](#)<sup>58</sup> ⌚ April 24, 2023 6:36 am

Since `#if 0` excludes Code, and `#if 1` includes Code, does this mean that somewhere, this '1' exists?

📝 Last edited 1 year ago by Aersuy

👍 0    ➡ Reply



**Alex** Author

🗨 Reply to [Aersuy](#)<sup>59</sup> ⌚ April 26, 2023 11:23 pm

Not in this context. If the expression (following the `#if`) evaluates to a nonzero value, the following code block is included. Otherwise it is skipped.

So in this case, 1 is an expression that evaluates to 1, which is non-zero, which will include the following code block.

The directives themselves (`#if`/`#endif`) are only of use to the preprocessor and are stripped out before compilation.

👍 0    ➡ Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. [https://en.cppreference.com/w/cpp/language/translation\\_phases](https://en.cppreference.com/w/cpp/language/translation_phases)
3. <https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/>
4. <https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/>
5. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#if0>
6. <https://www.learncpp.com/cpp-tutorial/header-files/>
7. <https://www.learncpp.com/>
8. <https://www.learncpp.com/introduction-to-the-preprocessor/>

9. <https://www.learncpp.com/cpp-tutorial/chapter-1-summary-and-quiz/>
10. <https://gravatar.com/>
11. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-595951>
12. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-596021>
13. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-595876>
14. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-594971>
15. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-594992>
16. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-593875>
17. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-592687>
18. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-592707>
19. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-590495>
20. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-589255>
21. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-588916>
22. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-588917>
23. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-587347>
24. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-587550>
25. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-587584>
26. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-586132>
27. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-586126>
28. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-585272>
29. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-582646>
30. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-582094>
31. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580807>
32. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580973>
33. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580516>
34. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580552>
35. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580330>
36. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580374>
37. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580443>
38. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580510>
39. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580308>
40. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580313>
41. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-586125>
42. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-586299>
43. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-588464>
44. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-579173>
45. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-579256>
46. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-578716>
47. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-578618>
48. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-578662>
49. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-577167>
50. <https://stackoverflow.com/questions/71671400/print-a-define-macro-using-stdcout>
51. <https://gcc.gnu.org/onlinedocs/gcc-3.4.3/cpp/Stringification.html>
52. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-577168>
53. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-577185>
54. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580808>
55. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-580974>

56. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-586128>
57. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-576953>
58. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-577081>
59. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/#comment-579786>